Extensions to Allocatables and Pointers

Hamid Oloso & Tom Clune

SIVO Fortran 2003 Series

February 26th 2008

Logistics

- Materials for this series can be found at http://modelingguru.nasa.gov/clearspace/docs/DOC-1375
  - Contains slides and source code examples.
  - Latest materials may only be ready at-the-last-minute.
- Please be courteous:
  - Remote attendees should use “*6” to toggle the mute. This will minimize background noise for other attendees.
- Webex - under investigation

Outline

- Introduction
- Standardized extensions to Fortran 95
  - Allocatable Dummy Arguments
  - Allocatable Function Results
  - Allocatable Components
- Allocatable entities
  - Allocatable Scalars
  - Assignment to an Allocatable Array
  - Transferring an Allocation
- Introduction to Typed and Sourced (Cloning) Allocation
- Pointer Assignment
- Procedure Pointers - deferred till OO
- Resources

Introduction

- Started as a standardized extension to Fortran 95 in Tech Report <Reference here> but now part of Fortran 2003:
  - Allocatable dummy arguments
  - Allocatable functions
  - Allocatable components
  - Pointers could be used but…..
    - Performance: cannot guarantee contiguous memory storage (stride 1)
    - Performance: aliasing (multiple refs to same entity) prevents some optimizations
    - Safety: can lead to subtle memory leaks and/or dangling pointers
- Additional extensions in Fortran 2003 proper:
  - Allocatable scalars
  - Assignment to an allocatable array
  - Transferring an allocation
  - Typed and Sourced (Cloning) Allocation - only brief intro here, more later under OO
  - Pointer assignment

Allocatable Dummy Arguments

- Dummy argument can have ALLOCATABLE attribute
- Corresponding actual argument must have same TKR and be ALLOCATABLE
- Allocation status
  - Dummy argument receives status of actual argument on entry
  - Actual argument receives status of dummy argument on return
  - Either way, status may be “not currently allocated”
- No reference to the associated actual argument is permitted via another alias if the dummy argument is allocated, deallocated, defined, or becomes undefined.
- “intent” permitted both for allocation status and array itself
  - intent(in) ⇒ array can not be allocated/deallocated and value can not be altered
  - Intent(out) ⇒ array allocated on entry becomes deallocated
  - Intent(inout) => array receives status from caller and sends status back to caller
- Example: Reading arrays of variable bounds

```
Subroutine load(array, unit)
   real, allocatable, intent(out),
   dimension(:,:,:) :: array
    integer, intent(in)     :: unit
    integer                      :: n1, n2, n3
    read(unit) n1, n2, n3
    allocate(array(n1, n2, n3))
    read(unit) array
End subroutine load
```

Allocatable Function Results

- Return value of a function can be allocatable, e.g.

```
FUNCTION af() RESULT(res)
  REAL, ALLOCATABLE :: res
```

- Allocation status of result on entry to function is “not currently allocated”
- Result may be allocated/deallocated any number of times during function execution
- Result must be allocated and have defined value on return from function
- Result is automatically deallocated after it has been used
  - *Important property which prevents memory leaks!*

Allocatable Function Results

- Example:

```
! The result of this function is the original argument with adjacent
! duplicate entries deleted (so if it was sorted, each element is unique).
FUNCTION compress(array)
      INTEGER,  ALLOCATABLE :: compress(:)
      INTEGER,  INTENT(IN) :: array(:)
      IF (SIZE(array,1)==0) THEN
            ALLOCATE(compress(0))
      ELSE
            N = 1
            DO I=2,SIZE(array,1)
                  IF (array(I)/=array(I-1)) N = N + 1
            END DO
            ALLOCATE(compress(N))
            N = 1
            compress(1) = array(1)
            DO I=2,SIZE(array,1)
                 IF (array(I)/=compress(N)) THEN
                      N = N + 1
                      compress(N) = array(I)
                 END IF
             END DO
      END IF
END
```

Allocatable components

- A structure component can be declared ALLOCATABLE:

```
TYPE t
  REAL, ALLOCATABLE :: c(:,:)
END TYPE
SUBROUTINE s()
  TYPE(t) x
  TYPE(t), SAVE :: y
  ...
END SUBROUTINE
```

  - As with variables, initially unallocated
    - x%c is unallocated upon each entry to subroutine s()
    - y%c is unallocated at the beginning of the program.
  - As with variables, automatically deallocated (unless SAVEd)
    - x%c is deallocated on return from subroutine s()
    - y%c retains its allocation status.

Allocatable components

- Unlike variables, sensible assignment ("deep copy")
- The assignment statement

  x = y

  acts like

```
IF (ALLOCATED(x%c)) DEALLOCATE(x%c)
IF (ALLOCATED(y%c)) THEN
   ALLOCATE(x%c(lbound(y%c,1):ubound(y%c,1), &
      lbound(y%c,2):ubound(y%c,2)))
   x%c = y%c
END IF
```

- This is recursively applied for nested allocatable components.
- Rationale: Otherwise the bookkeeping would be prohibitive.

Allocatable components

- Example

```
MODULE matrix_module
  TYPE real_matrix
    REAL,ALLOCATABLE :: value(:,:)
  END TYPE
  INTERFACE OPERATOR(*)
    MODULE PROCEDURE multiply_mm
  END INTERFACE
  ...
CONTAINS
  TYPE(real_matrix) FUNCTION multiply_mm(a,b) RESULT(c)
    TYPE(real_matrix),INTENT(IN) :: a,b
    ALLOCATE(c%value(size(a%value,1),size(b%value,2)))
    c%value = matmul(a%value,b%value)
  END FUNCTION
END

PROGRAM example
  USE matrix_module
  TYPE(real_matrix) :: x,y,z
  ...
  x = y*z
  ...
END
```

- Superior to version based upon pointers:
  - More efficient
  - No memory leak
  - Easier to write (e.g. assignment does the "right thing").

Allocatable scalars

- ALLOCATABLE attribute is now permitted for scalar variables/components
  - Particularly useful when combined with deferred type parameters

```
CHARACTER(:), ALLOCATABLE :: chdata
INTEGER :: unit, reclen
.
.
.
READ(unit) reclen
ALLOCATE(character(reclen) :: chdata)
READ(unit) chdata
```

- Automatically deallocated after use - prevents memory leaks

Assignment to allocatable arrays

- Fortran 95: an allocate variable must first be allocated in a separate statement before values are assigned to it in another statement
- Fotran 2003: allocation is automatic based on assignment
- Automatic allocation/reallocation for deferred type parameters as well
- Example:

Fortran 95

```
.
.
n = size(F(A))
  if (allocated(B)) then
      if (size(B) /= n) then
         deallocate(B)
         allocate(B(n))
      endif
   else
      allocate(B)
   end if
   B = F(A)
```

Fortran 2003

```
.
.
B = F(A)
```

Transferring an allocation

- Use intrinsic subroutine **move_alloc()**

  ```
  call move_alloc(from, to)
  ```
- **from** is allocatable and has intent **inout**
- **to** is allocatable of same **type** and **rank** as **from**
- After the call:
  - Original allocation of **to** is deallocated
  - New allocation status of **to** is that of **from**
  - **from** becomes deallocated
- Example:

```
real, allocatable :: a1(:), a2(:)
allocate (a1(0:10))
a1(3) =37
call move_alloc(from=a1, to=a2)
! a1 is now unallocated
! a2 is allocated with bounds (0:10) and a2(3) = 37.
```

Introduction to Typed and Sourced (Cloning) allocation

- The *`allocate`* statement can now determine:
  - Type parameter values (Type & Value)
- Controlled by either type specification in the *`allocate`* statement or by the use of *`source=`* clause
  - Syntax of the *`allocate`* statement is thus extended to:

    `allocate`([*type-spec* ::] *allocation-list* [,source=*source-expr*], [`stat`=*stat*])

    - *type-spec* is the type name followed by the type parameter values in parentheses
    - *source-expr* is any expression that is type-compatible
    - An `allocate` statement with a *type-spec* is *typed allocation*
    - An `allocate` statement with `source=` is a *sourced allocation*
    - Only one of *type-spec* or `source=` clauses is allowed in an `allocate` statement
  - Examples
    - *typed allocation*

      ```
      TYPE(matrix(KIND(0.0D0),m =:,n =:), ALLOCATABLE :: b,c
      ALLOCATE(TYPE(matrix(KIND(0.0D0),m = 10,n = 20)):: b,c )
      ```
    - *sourced allocation*

      ```
      TYPE(matrix(KIND(0.0D0),m = 10,n = 20):: a
      TYPE(matrix(KIND(0.0D0),m =:,n =:), ALLOCATABLE :: b
      ALLOCATE(b,SOURCE=a)
      ```

Pointer assignment

- **INTENT:** Controls changes to association status (not definition status).

```
SUBROUTINE pex(p1,p2,p3)
    .., POINTER, INTENT(IN) :: p1
    .., POINTER, INTENT(INOUT) :: p2
    .., POINTER, INTENT(OUT) :: p3
    ...
    p1 = 2    ! ok
    p1 => p2 ! not permitted
    p2 => p3 ! Permitted, but not safe
  END
```

- Notes:
  - p1 cannot have its association status altered during execution of pex(), except that it may become undefined if its target is deallocated (through some other pointer).
  - p2 and p3 must be associated with pointer variables, not pointer function references.
  - p3 has undefined association status on entry to pex().

Pointer assignment

- **Lower Bounds:** May be specified on pointer assignment.

```
REAL,POINTER :: a(:),b(:),c(:)
  ...
  ALLOCATE(a(-10:10))     ! Lower bound of A is -10
  b => a                  ! Lower bound of B is -10
  c => a(-5:5)            ! Lower bound of C is 1
  c(-5:) => a(-5:5)       ! Lower bound of C is -5
```

  - The upper bounds are derived from the specified lower bounds and the extent.

Pointer assignment

- **Rank Remapping:** Change rank in pointer assignment.
  - Motivation: allow different “views” to same region of memory
    - Use natural indexing for each algorithm
    - E.g. pointer to diagonal

```
REAL,ALLOCATABLE,TARGET :: base_array(:)
REAL,POINTER :: matrix(:,:)
REAL,POINTER :: diagonal(:)
...
ALLOCATE(base_array(n*n))
matrix(1:n,1:n) => base_array ! rank remapping
diagonal => base_array(::n+1)
```

- Notes:
  - The base array must be rank one, to ensure that the remapping is a simple linear transformation.
  - Both lower bound and upper bound must be specified for each dimension.

Pitfalls and Best Practices

- Best Practices
  - Use allocatables where appropriate instead of pointers
    - Efficiency
    - Convenience
    - Avoidance of memory leak - Fortran 2003 extensions automatically deallocate

Supported Features

Compiler	Ifort 9.1.049	Ifort 10.0.025	NAG 5.1	Xlf 11.0	G95 0.90	Gfortran 20070810	pgi 6.2.4
Allocatable Dummy Arguments	yes	yes	yes	yes	yes	no	no
Allocatable Function Results	yes	yes	yes	yes	yes	yes	no
Allocatable Components	yes	yes	yes	yes	yes	no	yes
Allocatable Scalars	no	no	no	yes	no	no	no
Assignment to an Allocatable Array	no	yes	no	no	no	no	no
Transfering an Allocation	yes	yes	no	yes	no	no	no
Pointer Lower Bound	no	yes	no	yes	no	no	no
Pointer Rank	no	yes	yes	yes	no	no	no

Feel free to contribute if you have access to other compilers not mentioned!

Resources

- **SIVO Fortran 2003 series:** https://modelingguru.nasa.gov/clearspace/docs/DOC-1390
- **Questions to Modeling Guru:** https://modelingguru.nasa.gov
- **SIVO code examples on Modeling Guru**
- **Fortran 2003 standard:** http://www.open-std.org/jtc1/sc22/open/n3661.pdf
- ***John Reid summary:***
  - ftp://ftp.nag.co.uk/sc22wg5/N1551-N1600/N1579.pdf
  - ftp://ftp.nag.co.uk/sc22wg5/N1551-N1600/N1579.ps.gz
- ***Newsgroups***
  - http://groups.google.com/group/comp.lang.fortran
- ***Real world examples***
  - ***Fortran 2003 Interface to OpenGL:*** http://www-stone.ch.cam.ac.uk/pub/f03gl/
  - ***Fotran 2003 version of NETCDF:*** ftp://ftp.unidata.ucar.edu/pub/netcdf/contrib/netcdf-3.6.1-f03-2.tgz
  - **FGSL: A Fortran interface to the GNU Scientific Library** http://www.lrz-muenchen.de/services/software/mathematik/gsl/fortran/index.html

Next Fortran 2003 Session

- I/O extensions
- Tom Clune will present
- Tuesday, March 11 2008
- B28-E210